Performance Programming with IBM pSeries Compilers

IBM SP Scientific Computing User Group

October 9, 2001

Bob Blainey
blainey@ca.ibm.com

Agenda

- **Review of the pSeries compiler products**
  - C for AIX, Version 5.0
  - VisualAge for C++ for AIX, Version 5.0
  - XL Fortran for AIX, Version 7.1
- **Tutorial on performance controls**
  - Performance compiler options
  - Directives and pragmas
- **Programming for performance**
- **A peek inside the compiler**
- **A close look at Power 4 optimization**
- **Q&A**

IBM Compiler Products for pSeries

- **Latest versions**
  - ►C for AIX, Version 5.0.2.0
  - ►VisualAge C++ Professional for AIX, Version 5.0.2.0
  - ►XL Fortran for AIX, Version 7.1.0.2
- **Older, supported versions**
  - ►XL High Performance Fortran for AIX, Version 1.4 (until 12/01)
  - ►VisualAge C++ Professional for AIX, Version 4.0 (until 12/02)

XL Fortran version 7.1

- Fortran 77/90/95 compiler with many extensions
- 32 and 64 bit support for serial and SMP
- OpenMP 1.0 support (OpenMP 2.0 coming ...)
- Support for TotalView, xldb, IBM distributed debugger and dbx/pdbx
- Snapshot directive for debugging optimized code
- Portfolio of optimizing transformations
  - ►Comprehensive path length reduction
  - ►Whole program analysis
  - ►Loop optimization for parallelism, locality and instruction scheduling
  - ►Tuned support for all RS/6000 and pSeries processors
- More info: www.software.ibm.com/ad/fortran

C for AIX version 5.0

- ANSI C89 compliant compiler (C99 coming soon)
- 32 and 64 bit support for serial and SMP
- Full support for OpenMP 1.0 (participating in OpenMP 2.0 definition)
- Support for TotalView, xldb, IBM distributed debugger and dbx/pdbx
- Snapshot directive for debugging optimized code
- Runtime memory debug support
- Portfolio of optimizing transformations
  - Similar to Fortran support but includes tuned optimizations for C pointers and systems coding styles
- More info: www.software.ibm.com/ad/caix

VisualAge for C++ for AIX version 5.0

- Fully compliant ANSI98 C++ compiler
- 32 and 64 bit support
- Batch compiler for traditional build environments and maximal optimization
- Incremental compiler for rapid application development (to be phased out in next release)
- Integrated graphical development environment including remote debug and performance visualization
- Support for TotalView, xldb, IBM distributed debugger and dbx/pdbx
- Portfolio of optimizing transformations
  - Subset of transformations available in Fortran and C but has tuned support for all processors
  - Much more coming soon
- More info: www.software.ibm.com/ad/vacpp

Performance Compiler Options

- Optimization level
- High order transformations
- Interprocedural analysis
- Profile directed feedback
- Target machine specification
- Floating point options
- Program behaviour
- Diagnostic options

Optimization Level

- **OPTIMIZE: specified as -qoptimize=n or -On where n is one of:**
  - **0**: Fast compilation, full support for debugging
  - **2**: Comprehensive low-level optimization, partial support for debugging (procedure boundaries)
  - **3**: Even more optimization - compile time/space intensive and/or marginal effectiveness
  - **4**: Macro option including -O3, -qhot, -qipa, -qarch=auto, -qtune=auto, -qcache=auto
  - **5**: Macro option including -O4, -qipa=level=2

Optimization Options (*continued*)

- **Examples of optimizations done at -O or -O2**
  - Global assignment of user variables to registers
  - Effective usage of addressing modes (eg. update)
  - Elimination of unused or redundant code
  - Movement of invariant code out of loops
  - Scheduling of instructions for the target machine
  - Some loop unrolling and scheduling
- **Examples of optimizations done at -O3**
  - Deeper inner loop unrolling
  - Better loop scheduling
  - Additional optimizations allowed by -qnostrict
  - Widened optimization scope (typically whole procedure)
  - No implicit memory usage limits (-qmaxmem=-1)

Example: Matrix Multiply

```
DO I = 1, N1
  DO J = 1, N3
    Z(I,J) = 0.0
    DO K = 1, N2
      Z(I,J) = Z(I,J) + X(I,K) * Y(K,J)
    END DO
  END DO
END DO
```

Matrix multiply with no optimization

```
13|                               CL.4:
14| 000180 lwz      809F0000   1      L4A      gr4=i(gr31,0)
14| 000184 lwz      807F0004   0      L4A      gr3=j(gr31,4)
14| 000188 addi     3903FFFF   2      AI       gr8=gr3,-1
14| 00018C lwz      80BF0024   0      L4A      gr5=#13(gr31,36)
14| 000190 lwz      806100A0   1      L4A      gr3=.z(gr1,160)
14| 000194 rlwinm   54841838   0      SLL4     gr4=gr4,3
14| 000198 mullw    7CA829D6   2      M        gr5=gr8,gr5,mq"
14| 00019C add      7CC42A14   1      A        gr6=gr4,gr5
14| 0001A0 add      7CC33214   0      A        gr6=gr3,gr6
14| 0001A4 lfd      C826FFF8   1      LFL      fp1=z(gr6,-8)
14| 0001A8 lwz      80FF0008   0      L4A      gr7=k(gr31,8)
14| 0001AC addi     3927FFFF   2      AI       gr9=gr7,-1
14| 0001B0 lwz      815F000C   0      L4A      gr10=#7(gr31,12)
14| 0001B4 lwz      80C10098   1      L4A      gr6=.x(gr1,152)
14| 0001B8 mullw    7D2951D6   2      M        gr9=gr9,gr10,mq"
14| 0001BC add      7D244A14   1      A        gr9=gr4,gr9
14| 0001C0 add      7CC64A14   0      A        gr6=gr6,gr9
14| 0001C4 lfd      C846FFF8   1      LFL      fp2=x(gr6,-8)
14| 0001C8 lwz      813F0018   0      L4A      gr9=#10(gr31,24)
14| 0001CC lwz      80C1009C   1      L4A      gr6=.y(gr1,156)
14| 0001D0 rlwinm   54E71838   0      SLL4     gr7=gr7,3
14| 0001D4 mullw    7D0849D6   2      M        gr8=gr8,gr9,mq"
14| 0001D8 add      7CE74214   1      A        gr7=gr7,gr8
14| 0001DC add      7CC63A14   0      A        gr6=gr6,gr7
14| 0001E0 lfd      C866FFF8   1      LFL      fp3=y(gr6,-8)
14| 0001E4 fmadd    FC2208FA   1      FMA      fp1=fp1-fp3,fcr
14| 0001E8 add      7C842A14   0      A        gr4=gr4,gr5
14| 0001EC add      7C632214   0      A        gr3=gr3,gr4
14| 0001F0 stfd     D823FFF8   0      STFL     z(gr3,-8)=fp1
15| 0001F4 lwz      807F0008   1      L4A      gr3=k(gr31,8)
15| 0001F8 addi     38630001   2      AI       gr3=gr3,1
15| 0001FC stw      907F0008   1      ST4A     k(gr31,8)=gr3
15| 000200 lwz      80610070   0      L4A      gr3=#21(gr1,112)
15| 000204 addic.   3463FFFF   2      AI_R     gr3=gr3,-1
15| 000208 stw      90610070   0      ST4A     #21(gr1,112)=gr3
15| 00020C bc       4181FF74   1      BT       CL.4,cr0,0x2/gt ,
```

Matrix multiply with -O2

```
14| 000094 lfd      C83F0008   1     LFL      fp1=y(gr31,8)
14| 000098 lfdux    7C5E3CEE   1     LFDU     fp2,gr30=x(gr30,gr7,0)
14| 00009C lfd      C87F0010   1     LFL      fp3=y(gr31,16)
14| 0000A0 lfdux    7C9E3CEE   1     LFDU     fp4,gr30=x(gr30,gr7,0)
14| 0000A4 lfd      C8BF0018   1     LFL      fp5=y(gr31,24)
14| 0000A8 lfdux    7CDE3CEE   1     LFDU     fp6,gr30=x(gr30,gr7,0)
14| 0000AC lfdu     CD1F0020   1     LFDU     fp8,gr31=y(gr31,32)
 0| 0000B0 bc       43400038   0     BCF      ctr=CL.101,taken=0%(0,100)
13|                              CL.4:
14| 0000B4 fmadd    FCE0387A   1     FMA      fp7=fp7,fp0,fp1,fcr
14| 0000B8 lfdux    7C1E3CEE   1     LFDU     fp0,gr30=x(gr30,gr7,0)
14| 0000BC lfd      C83F0008   1     LFL      fp1=y(gr31,8)
14| 0000C0 fmadd    FCE238FA   2     FMA      fp7=fp7,fp2,fp3,fcr
14| 0000C4 lfdux    7C5E3CEE   0     LFDU     fp2,gr30=x(gr30,gr7,0)
14| 0000C8 lfd      C87F0010   1     LFL      fp3=y(gr31,16)
14| 0000CC fmadd    FCE4397A   3     FMA      fp7=fp7,fp4,fp5,fcr
14| 0000D0 lfdux    7C9E3CEE   0     LFDU     fp4,gr30=x(gr30,gr7,0)
14| 0000D4 lfd      C8BF0018   0     LFL      fp5=y(gr31,24)
14| 0000D8 fmadd    FCE63A3A   4     FMA      fp7=fp7,fp6,fp8,fcr
14| 0000DC lfdu     CD1F0020   0     LFDU     fp8,gr31=y(gr31,32)
14| 0000E0 lfdux    7CDE3CEE   0     LFDU     fp6,gr30=x(gr30,gr7,0)
 0| 0000E4 bc       4320FFD0   0     BCT      ctr=CL.4,taken=100%(100,0)
 0|                              CL.101:
14| 0000E8 fmadd    FC00387A   1     FMA      fp0=fp7,fp0,fp1,fcr
14| 0000EC fmadd    FC0200FA   4     FMA      fp0=fp0,fp2,fp3,fcr
14| 0000F0 fmadd    FC04017A   4     FMA      fp0=fp0,fp4,fp5,fcr
14| 0000F4 fmadd    FCE6023A   4     FMA      fp7=fp0,fp6,fp8,fcr
```

Matrix multiply with -O3

```
14| 00009C lfdux    7C3E3CEE   1     LFDU    fp1,gr30=x(gr30,gr7,0)
14| 0000A0 lfdux    7C5E3CEE   1     LFDU    fp2,gr30=x(gr30,gr7,0)
14| 0000A4 lfd      C87F0008   1     LFL     fp3=y(gr31,8)
14| 0000A8 lfd      C89F0010   1     LFL     fp4=y(gr31,16)
 0| 0000AC lfs      C0FD0000   1     LFS     fp7=+CONSTANT_AREA(gr29,0)
14| 0000B0 lfdux    7CBE3CEE   1     LFDU    fp5,gr30=x(gr30,gr7,0)
14| 0000B4 lfd      C8DF0018   1     LFL     fp6=y(gr31,24)
 0| 0000B8 fmr      FD003890   1     LRFL    fp8=fp7
 0| 0000BC fmr      FD603890   1     LRFL    fp11=fp7
 0| 0000C0 bc       43400038   0     BCF     ctr=CL.110,taken=0%(0,100)
13|                               CL.4:
14| 0000C4 fmadd    FC0100FA   1     FMA     fp0=fp0,fp1,fp3,fcr
14| 0000C8 lfdux    7D3E3CEE   1     LFDU    fp9,gr30=x(gr30,gr7,0)
14| 0000CC fmadd    FCE2393A   1     FMA     fp7=fp7,fp2,fp4,fcr
14| 0000D0 lfdu     CD5F0020   1     LFDU    fp10,gr31=y(gr31,32)
14| 0000D4 fmadd    FD0541BA   1     FMA     fp8=fp8,fp5,fp6,fcr
14| 0000D8 lfdux    7C3E3CEE   1     LFDU    fp1,gr30=x(gr30,gr7,0)
14| 0000DC lfd      C87F0008   1     LFL     fp3=y(gr31,8)
14| 0000E0 lfdux    7C5E3CEE   1     LFDU    fp2,gr30=x(gr30,gr7,0)
14| 0000E4 lfd      C89F0010   1     LFL     fp4=y(gr31,16)
14| 0000E8 fmadd    FD695ABA   1     FMA     fp11=fp11,fp9,fp10,fcr
14| 0000EC lfdux    7CBE3CEE   1     LFDU    fp5,gr30=x(gr30,gr7,0)
14| 0000F0 lfd      C8DF0018   1     LFL     fp6=y(gr31,24)
 0| 0000F4 bc       4320FFD0   0     BCT     ctr=CL.4,taken=100%(100,0)
 0|                               CL.110:
14| 0000F8 fmadd    FC0100FA   1     FMA     fp0=fp0,fp1,fp3,fcr
14| 0000FC lfdu     CC3F0020   1     LFDU    fp1,gr31=y(gr31,32)
14| 000100 fmadd    FC42393A   1     FMA     fp2=fp7,fp2,fp4,fcr
14| 000104 lfdux    7C7E3CEE   1     LFDU    fp3,gr30=x(gr30,gr7,0)
14| 000108 fmadd    FC8541BA   1     FMA     fp4=fp8,fp5,fp6,fcr
14| 00010C fmadd    FC23587A   1     FMA     fp1=fp11,fp3,fp1,fcr
 0| 000110 fadd     FC00102A   1     AFL     fp0=fp0,fp2,fcr
 0| 000114 fadd     FC24082A   3     AFL     fp1=fp4,fp1,fcr
 0| 000118 fadd     FC00082A   4     AFL     fp0=fp0,fp1,fcr
```

Tips for getting the most out of -O2/3

- If possible, test and debug your code without optimization before using -O2 or -O3
- Ensure that your code is standard-compliant. Optimizers are the ultimate conformance test!
  - In Fortran code, ensure that subroutine parameters comply with aliasing rules
  - In C code, ensure that pointer use follows type restrictions
  - Ensure all shared variables are marked volatile
- Compile as much of your code as possible with -O2.
- If you encounter problems with -O2, consider using -qalias=noansi or -qalias=nostd rather that turning off optimization.
- Next, use -O3 on as much code as possible.
- If you encounter problems or degradations, consider using -qstrict or -qcompact along with -O3 where necessary.
- If you still have problems with -O3, switch to -O2 for a subset of files/subroutines but consider using -qmaxmem=-1 and/or -qnostrict.

Optimization Options (*continued*)

- **HOT (High Order Transformations) - Fortran (C and C++ coming soon)**
  - Specified as -qhot[=[no]vector | arraypad[=*n*]]
  - Optimized handling of F90 array language constructs (elimination of temporaries, fusion of statements)
  - High level transformation (eg. interchange) of loop nests to improve memory locality (reduce cache/TLB misses), optimize usage of hardware prefetch and balance loop computation (typically ld/st vs. float)
  - *Optionally* transforms loops to exploit vector intrinsic library (eg. reciprocal, sqrt, trig) - may result in slightly different rounding
  - *Optionally* introduces array padding under user control - potentially unsafe if not applied uniformly

Matrix multiply with -O3 -qhot

```
 13|                                  CL.4:
 14| 0001C8 fmadd    FC0200FA   1      FMA      fp0=fp0,fp2,fp3,fcr
 14| 0001CC lfdux    7FDA34EE   1      LFDU     fp30,gr26=x(gr26,gr6,0)
 14| 0001D0 fmadd    FF42D1FA   1      FMA      fp26=fp26,fp2,fp7,fcr
 14| 0001D4 lfdu     CFF80010   1      LFDU     fp31,gr24=y(gr24,16)
 14| 0001D8 fmadd    FF64D9FA   1      FMA      fp27=fp27,fp4,fp7,fcr
 14| 0001DC lfdu     CFB90010   1      LFDU     fp29,gr25=y(gr25,16)
 14| 0001E0 fmadd    FC23093A   1      FMA      fp1=fp1,fp3,fp4,fcr
 14| 0001E4 lfd      CB9A0008   1      LFL      fp28=x(gr26,8)
 14| 0001E8 lfd      C8780008   1      LFL      fp3=y(gr24,8)
 14| 0001EC lfdux    7C5A34EE   1      LFDU     fp2,gr26=x(gr26,gr6,0)
 14| 0001F0 lfd      C89A0008   1      LFL      fp4=x(gr26,8)
 14| 0001F4 lfd      C8F90008   1      LFL      fp7=y(gr25,8)
 14| 0001F8 fmadd    FF45D27A   1      FMA      fp26=fp26,fp5,fp9,fcr
 14| 0001FC fmadd    FC0501BA   1      FMA      fp0=fp0,fp5,fp6,fcr
 14| 000200 lfdux    7CBA34EE   1      LFDU     fp5,gr26=x(gr26,gr6,0)
 14| 000204 fmadd    FC260A3A   1      FMA      fp1=fp1,fp6,fp8,fcr
 14| 000208 lfdu     CCD80010   1      LFDU     fp6,gr24=y(gr24,16)
 14| 00020C fmadd    FF68DA7A   1      FMA      fp27=fp27,fp8,fp9,fcr
 14| 000210 lfdu     CD390010   1      LFDU     fp9,gr25=y(gr25,16)
 14| 000214 lfd      C91A0008   1      LFL      fp8=x(gr26,8)
 14| 000218 fmadd    FC0B02BA   1      FMA      fp0=fp0,fp11,fp10,fcr
 14| 00021C fmadd    FF4BD37A   1      FMA      fp26=fp26,fp11,fp13,fcr
 14| 000220 lfdux    7D7A34EE   1      LFDU     fp11,gr26=x(gr26,gr6,0)
 14| 000224 fmadd    FF6CDB7A   1      FMA      fp27=fp27,fp12,fp13,fcr
 14| 000228 lfd      C9B90008   1      LFL      fp13=y(gr25,8)
 14| 00022C fmadd    FC2A0B3A   1      FMA      fp1=fp1,fp10,fp12,fcr
 14| 000230 lfd      C9580008   1      LFL      fp10=y(gr24,8)
 14| 000234 lfd      C99A0008   1      LFL      fp12=x(gr26,8)
 14| 000238 fmadd    FF5ED77A   1      FMA      fp26=fp26,fp30,fp29,fcr
 14| 00023C fmadd    FC1E07FA   1      FMA      fp0=fp0,fp30,fp31,fcr
 14| 000240 fmadd    FC3F0F3A   1      FMA      fp1=fp1,fp31,fp28,fcr
 14| 000244 fmadd    FF7CDF7A   1      FMA      fp27=fp27-fp29,fcr
  0| 000248 bc       4320FF80   0      BCT      ctr=CL.4,taken=100%(100,0)
```

Vectorization Example

```
SUBROUTINE VD(A,B,C,N)
REAL*8 A(N),B(N),C(N)
DO I = 1, N
  A(I) = C(I) / SQRT(B(I))
END DO
END
```

```
               SUBROUTINE vd (a, b, c, n)
                 @ICMO = n
    3|           IF ((@ICMO > 0)) THEN
    4|             @NumElements0 = int(int(@ICMO))
                    CALL __vrsqrt_630((a + (-8) + (8)*(1)),(b + (-8) + (8)*(1)),
              &     @NumElements0)
    3|             @CIVO = 0
      Id=3         DO @CIVO = @CIVO, int(@ICMO)-1
    4|               a((@CIVO + 1)) = c((@CIVO + 1)) * a((@CIVO + 1))
    5|             ENDDO
                 ENDIF
    6|           RETURN
               END SUBROUTINE vd
```

Tips for getting the most out of -qhot

- Try using -qhot along with -O2 or -O3 for all of your code. It is designed to have neutral effect when no opportunities exist.
- If you encounter unacceptably long compile times (this can happen with complex loop nests) or if your performance degrades with the use of -qhot, try using -qhot=novector, or -qstrict or -qcompact along with -qhot.
- If possible, report long compile times or poor generated code to IBM through your service representative. If that doesn't work, feel free to contact me.
- If necessary, deactivate -qhot selectively, allowing it to improve some of your code.
- Read the transformation report generated using -qreport (Fortran only for now). If your hot loops are not transformed as you expect, try using assertive directives such as INDEPENDENT or CNCALL or prescriptive directives such as UNROLL or PREFETCH.

Optimization Options (*continued*)

- **IPA (Inter-Procedural Analysis) - Fortran and C (C++ coming soon)**
  - Specified as -qipa[=level=*n* | inline= | *fine tuning*] on both compile *and* link steps
  - Expand the scope of optimization to an entire program unit (executable or shared object)
  - *level=0*: Program partitioning and simple interprocedural optimization
  - *level=1*: Inlining and global data mapping
  - *level=2*: Global alias analysis, specialization, interprocedural data flow
  - *inline=*: Precise user control of inlining
  - *fine tuning*: Specify library code behaviour, tune program partitioning, read commands from a file

IPA in depth

- level=0
  - automatic recognition of standard libraries
  - localization of statically bound variables and procedures
  - partitioning and layout of code according to call affinity
    - expansion of backend optimizer scope
- level=1
  - procedure inlining
  - partitioning and layout of static data according to reference affinity
- level=2
  - whole program alias analysis
  - aggressive intraprocedural optimizations
    - value numbering, code propagation and simplification, code motion (into conditions, out of loops), redundancy elimination
  - interprocedural constant propagation, dead code elimination, pointer analysis
  - procedure specialization (cloning)

Tips for getting the most from -qipa

- When specifying optimization options in a makefile, remember to repeat all options on the link step
  - OPT = -O3 -qipa
  - FFLAGS=...$(OPT)...
  - LDFLAGS=...$(OPT)...
- -qipa works when building executables or shared objects but always compile 'main' and exports with -qipa.
- It is not necessary to compile everything with -qipa but try to apply it to as much of your program as possible.
- When compiling and linking separately, use -qipa=noobject on the compile step for faster compilation.
- Ensure there is enough space in /tmp (at least 200MB) or use the TMP_DIR variable to specify a different directory.
- The "level" suboption is a throttle. Try varying the "level" suboption if compilation time is too long. -qipa=level=0 can be very beneficial for little cost.
- Look at the generated code. If too few or too many functions are inlined, consider using -qipa=[no]inline

Optimization Options (*continued*)

- **PDF (Profile-Directed Feedback)**: specified as -qpdf1 and -qpdf2
  - *-qpdf1* causes the resulting object to be instrumented for the collection of program control flow data
  - *-qpdf2* causes the compiler to consume previously collected data for the purpose of path-biased optimization
    - code layout, scheduling, register allocation
    - (in XLF 7.1.1, C/C++ V6) inlining decisions, partially invariant code motion, switch code generation, loop optimizations
  - Three step process:
    - Compile/link with -qpdf1
    - Run program through sample data
    - Compile/link with -qpdf2
      - (in XLF 7.1.1, C/C++ V6) only need to relink with -qpdf2.
  - PDF should be used mainly on code which has rarely executed conditional error handling or instrumentation
  - PDF usually has a neutral effect in the absence of firm profile information (ie. when sample data is inconclusive)
  - However, always use characteristic data for profiling. If sufficient data is unavailable, do not use PDF.

Optimization Options (*continued*)

- **COMPACT**: specified as -q[no]compact
  - Prefers final code size reduction over execution time performance when a choice is necessary
- **INLINE**: specified as -Q[+*names* | -*names* | !]
  - Controls inlining of named functions - usable at compile time and/or link time
- **UNROLL**: specified as -q[no]unroll
  - Independently controls loop unrolling (implicitly activated under -O2 and -O3)

Optimization Options (*continued*)

- **INLGLUE** - Specified as *-q[no]inlglue*
  - Inline calls to "glue" code used in calls through function pointers (including *virtual*) and calls to functions which are dynamically bound
  - Pointer glue is inlined by default for -qtune=pwr4
- **TBTABLE**
  - Controls the generation of traceback table information:
  - *-qtbtable=none* inhibits generation of tables - no stack unwinding is possible
  - *-qtbtable=small* generates tables which allow stack unwinding but omit name and parameter information - useful for optimized C++
    - This is the default setting when using optimization
  - *-qtbtable=full* generates full tables including name and parameter information - useful for debugging

Target Machine Options

- **ARCH**
  - Restricts the compiler to generate a subset of the Power or PowerPC instruction set
  - Specified as -qarch=*isa* where *isa* is one of:
    - *com* (default): Code can run on any RS/6000 - implies -qtune=pwr2
    - *auto*: Code may take advantage of instructions available only on the <u>compiling</u> machine (or similar machines)
    - *ppc*: Code follows PowerPC architecture - implies -qtune=604 (32 bit) or -qtune=pwr3 (64 bit)
    - *pwr3*: Code can run on any Power 3 - implies -qtune=pwr3
    - Lots of others: *pwr, pwr2, 604, pwr4, ...*

Target Machine Options *(continued)*

- **TUNE:** Bias optimization toward execution on a given machine
  - Does *not* imply anything about the ability to run correctly on a given machine - only affects performance
  - -qtune=auto generates code that is automatically tuned for the <u>compiling</u> machine (or similar machines)
  - Specified as -qtune=*machine* where *machine* is one of auto, 604, pwr2, p2sc, pwr3, pwr4, rs64c, etc.
- **CACHE**: Defines a specific cache/memory geometry
  - Defaults are set through TUNE
  - Specified as -qcache=level=*n*:*cache_spec*, where *cache_spec* includes:
    - type=i|d|c: cache type (instruction/data/combined)
    - line=*lsz*:size=*sz*:assoc=*as*: line/cache size and set associativity
    - cost=*c*: cost (in cpu cycles) of a miss
  - Mainly useful when using -qhot or -qsmp

Getting the most out of ARCH, TUNE and CACHE

- Try to specify with ARCH the narrowest family of machines possible that will be expected to run your code <u>correctly</u>.
  - -qarch=com will generate code that runs anywhere but will have slower integer divides and multiplies and will be unable to exploit single precision floating point
  - -qarch=ppc is better if you don't need to run on Power or Power2 but this will inhibit generation of sqrt or fsel, for example
  - -qarch=ppcgr is a bit better, since it allows generation of fsel but still no sqrt
  - To get sqrt, you will need -qarch=pwr3. This will also generate correct code for Power 4.
- Try to specify with TUNE the machine where performance should be best.
  - If you are not sure, try -qtune=pwr3. This will generate code that should generally run well on most machines.
- Before using the CACHE option, have a look at the options sections of the listing to see if the current settings are satisfactory. If you do decide to use -qcache, use -qhot along with it.

Target Machine Options *(continued)*

- **64/32**: Generate code for 64 bit (4/8/8) or 32 bit (4/4/4) addressing model
  - Specified as -q32 or -q64
  - -q64 generates code with different magic numbers on AIX V4 and AIX V5. If you code needs to run on both, build two executables or two libraries.
- **SMP (Fortran, C)**: Generate threaded code for a shared-memory parallel machine
  - Specified as -qsmp[=[no]auto:[no]omp:[no]opt:*fine tuning*]
  - *auto* instructs the compiler to automatically generate parallel code where possible without user assistance
  - *omp* instructs the compiler to observe OpenMP 1.0 language extensions for specifying explicit parallelism
  - *opt* instructs the compiler to optimize as well as parallelize. The optimization is equivalent to -O2 -qhot by default. The default setting is -qsmp=opt.
  - *fine tuning* includes control over thread scheduling, nested parallelism and locking

Getting the most out of -qsmp

- Test your programs using optimization and preferably using -qhot in a single-threaded manner before using -qsmp (where practical).
- Always use the "_r" or reentrant compiler invocations when using -qsmp.
- By default, the runtime will use all available processors. Do not set the PARTHDS or OMP_NUM_THREADS variables unless you wish to use fewer than the number of available processors.
- If using a machine or node in a dedicated fashion, consider setting the SPINS and YIELDS environment variables to 0.
- When debugging an OpenMP program, try using -qsmp=noopt (without -O) to make debugging information produced from the compiler more precise.

Floating Point Options

- **FLOAT**
  - Precise control over the handling of floating point calculations
  - Specified as -qfloat=*subopt* where *subopt* is one of:
    - *[no]fold*: enable compile time evaluation of floating point calculations - may want to disable for handling of certain exceptions (eg. overflow, imprecise)
    - *[no]maf*: enable generation of multiple-add type instructions - may want to disable for <u>exact</u> compatibility with other machines but this will come at a high price in performance
    - *[no]rrm*: specifies that rounding mode may not be round-to-nearest (default is *norrm*)

Floating Point Options (*continued*)

- **FLOAT** (*continued*)
  - *[no]hsflt*: allow various fast floating point optimizations including replacement of division by multiplication by a reciprocal
  - *[no]rsqrt*: allow computation of a divide by square root to be replaced by a multiply of the reciprocal square root
- **FLTTRAP**
  - Enables software-only checking of IEEE floating point exceptions
  - Usually more efficient than hardware checking since checks can be executed less frequently
  - Specified as -qflttrap=imprecise | enable | *ieee_exceptions*

Program Behaviour Options

- **STRICT**
  - Specified as -q[no]strict, default is -qstrict with -qoptimize=0 and -qoptimize=2, -qnostrict with -qoptimize=3,4,5
  - *nostrict* allows the compiler to reorder floating point calculations and potentially excepting instructions
- **ALIAS (Fortran)**
  - Specified as -qalias=[no]std:[no]aryovrlp:*others*
  - Allows the compiler to assume that certain variables do not refer to overlapping storage
  - *std* (default) refers to the rule about storage association of reference parameters with each other and globals
  - *aryovrlp* (default) defines whether there are any assignments between storage-associated arrays - try -qalias=noaryovrlp for better performance

Program Behaviour Options (*continued*)

- **ALIAS (C, C++)**
  - Similar to Fortran option of the same name but focussed on overlap of storage accessed using pointers
  - Specified as -qalias=*subopt* where *subopt* is one of:
    - *[no]ansi*: Enable ANSI standard type-based alias rules
    - *[no]typeptr*: Assume pointers to different types <u>never</u> point to the same or overlapping storage
    - *[no]allptrs*: Assume that different pointer variables always point to non-overlapping storage
    - *[no]addrtaken*: Assume that external variables do not have their address taken outside the source file being compiled

Why the big fuss about aliasing?

- The precision of compiler analyses is gated in large part by the apparent effects of direct or indirect memory writes and the apparent presence of direct or indirect memory reads.
- Memory can be referenced directly through a named symbol, indirectly through a pointer or reference parameter, or indirectly through a function call.
- Many apparent references to memory are false and these constitute barriers to compiler analysis.
- The compiler does analysis of possible aliases at all optimization levels but analysis of these apparent references is best when using -qipa since it can see through most calls.
- Options such as -qalias and directives such as disjoint, isolated_call, CNCALL and INDEPENDENT can have pervasive effect since they fundamentally improve the precision of compiler analysis.

Program Behaviour Options (*continued*)

- **ASSERT (Fortran, C)**
  - Specified as -qassert=[no]deps | itercnt=*n*
  - *deps* (default) indicates that some loop has a loop carried memory dependence - try -qassert=nodeps for improved performance
  - *itercnt* modifies the default assumptions about the expected iteration count of loops (normally 10)
- **INTSIZE (Fortran)**: Define the default size of INTEGER variables
  - Specified as -qintsize=1|2|4|8
  - When using -q64, try -qintsize=8 for improved performance
- **IGNERRNO (C,C++)** - Specified as *-q[no]ignerrno*
  - Indicates that the value of *errno* is not needed by the program
  - Can help in optimization of math functions.
  - This is the default with -O3.

Program Behaviour Options (*continued*)

- **DATA/PROC LOCAL/IMPORTED -** Specifies expected access to external variables and functions:
  - *-qdatalocal[=vars]*: Specifies that the definitions of all or just the named variables will be statically bound - access to statically bound variables is faster
  - *-qdataimported[=vars]*: Specifies that the definitions of all or just the named variables might be dynamically bound
  - *-qproclocal[=funcs]*: Specifies that the definitions of all or just the named functions will be statically bound - calls to statically bound functions are faster than dynamic or unknown
  - *-qprocimported[=funcs]*: Specifies that the definitions of all or just the named functions will be dynamically bound
  - *-qprocunknown[=funcs]*: Specifies that the definitions of all or just the named functions have unknown linkage

Program Behaviour Options (*continued*)

- **LIBANSI** (C, C++) - Specified as *-q[no]libansi*
  - Specifies that calls to ANSI standard functions will be bound with conforming implementations
  - This is the default with -qipa.
- **MA (C, C++)** - Specified as *-qma*
  - Directs the compiler to generate inline code for calls to the *alloca* function.
- **PROTO (C)** - Specified as *-q[no]proto*
  - Asserts that procedure call points agree with their declarations even if the procedure has not been prototyped.
  - Useful for well behaved K&R C code.
- **RO,ROCONST (C,C++)** - Specified as *-q[no]ro{const}*
  - Directs the compiler to place string literals (RO) or constant values (ROCONST) in read-only storage

Diagnostic Options

- **LIST**
  - Specified as -qlist
  - Instructs the compiler to emit an object listing
  - The object listing includes hex and pseudo-assembly representations of the generated code along with traceback tables and text constants
- **REPORT (Fortran)**
  - Specified as -qreport [=smplist]
  - Instructs the high level optimizer to emit a report including pseudo-Fortran along with annotations describing what transformations were performed (eg. loop unrolling, automatic parallelization)
  - Also includes information about data dependences and other inhibitors to optimization

Diagnostic Options (*continued*)

- **INITAUTO**
  - Directs the compiler to emit code that initializes all automatic (stack) variables to a given value
  - *-qinitauto=XX* initializes bytes with the value given in hex
  - *-qinitauto=XXXXXXXX* initializes words with the value given in hex

Directives and Pragmas

- **OpenMP 1.0** - supported in C and Fortran
- **Legacy SMP** directives and pragmas
  - Most of these are superceded by OpenMP - use OpenMP where possible
- **Assertive directives** (Fortran)
  - ASSERT, INDEPENDENT, CNCALL, PERMUTATION
- **Assertive pragmas** (C)
  - *isolated_call*, *disjoint, independent_loop, independent_calls, iterations, permutation, execution_frequency, leaves*
- **Embedded Options**
  - *#pragma options* and *#pragma option_override* in C
  - @PROCESS in Fortran
- **Prescriptive directives** (Fortran)
  - PREFETCH, UNROLL
- **Prescriptive pragmas** (C)
  - *sequential_loop*

Assertive Directives (Fortran)

- **ASSERT** (ITERCNT(*n*) | [NO]DEPS)
  - Same as options of the same name but applicable to a single loop - much more useful
- **INDEPENDENT:** Asserts that the following loop has *no* loop carried dependences - enables locality and parallel transformations
- **CNCALL:** Asserts that the calls in the following loop do not cause loop carried dependences
- **PERMUTATION** (*names*)
  - Asserts that elements of the named arrays take on distinct values on each iteration of the following loop - may be useful in sparse codes

Assertive Pragmas (C)

- *isolated_call* (*function_list*) asserts that calls to the named functions do not have side effects
- *disjoint* (*variable_list*) asserts that none of the named variables share overlapping areas of storage
- *independent_loop* is equivalent to INDEPENDENT
- *independent_calls* is equivalent to CNCALL
- *permutation* is equivalent to PERMUTATION
- *iterations* is equivalent to ASSERT(ITERCNT)
- *execution_frequency* (*very_low*) asserts that the control path containing the pragma will be infrequently executed
- *leaves* (*function_list*) asserts that calls to the named functions will not return (eg. exit)

Prescriptive Directives (Fortran)

- **PREFETCH**
  - PREFETCH_BY_LOAD (*variable_list*): issue *dummy* loads to cause the given variables to be prefetched into cache - useful on Power machines or to activate Power 3 hardware prefetch
  - PREFETCH_FOR_LOAD (*variable_list*): issue a *dcbt* instruction for each of the given variables.
  - PREFETCH_FOR_STORE (*variable_list*): issue a *dcbtst* instruction for each of the given variables.
- **UNROLL**
  - Specified as [NO]UNROLL [(*n*)]
  - Used to activate/deactivate compiler unrolling for the following loop.
  - Can be used to give a specific unroll factor.

Prescriptive Pragmas (C)

- *sequential_loop* directs the compiler to execute the following loop in a single thread, even if the -qsmp=auto option is specified

Compiler Friendly Programming

- Compiler-friendly programming idioms can be as useful to performance as any of the options or directives
- Do not excessively hand-optimize your code (eg. unrolling, inlining) - this often confuses the compiler (and other programmers!) and makes it difficult to optimize for new machines
- Avoid unnecessary use of globals and pointers - when using them in a loop, load them into a local before the loop and store them back after.
- Avoid breaking your program into too many small functions. If you must use small functions, seriously consider using -qipa.
- Use register-sized integers (**long** in C/C++ and **INTEGER*4** or **INTEGER*8** in Fortran) for scalars. For large arrays of integers, consider using 1 or 2 byte integers or bitfields in C or C++.

Compiler Friendly Programming *(continued)*

- Use the smallest floating point precision appropriate to your computation. Use 'long double', 'REAL*16' or 'COMPLEX*32' only when extremely high precision is required.
- Obey all language aliasing rules (try to avoid -qassert=nostd in Fortran and -qalias=noansi in C/C++)
- Use locals wherever possible for loop index variables and bounds. In C/C++, avoid taking the address of loop indices and bounds.
- Keep array index expressions as simple as possible. Where indexing needs to be indirect, consider using the PERMUTATION directive.
- Consider using the highly tuned MASS and ESSL libraries rather than custom implementations or generic libraries

Fortran programming tips

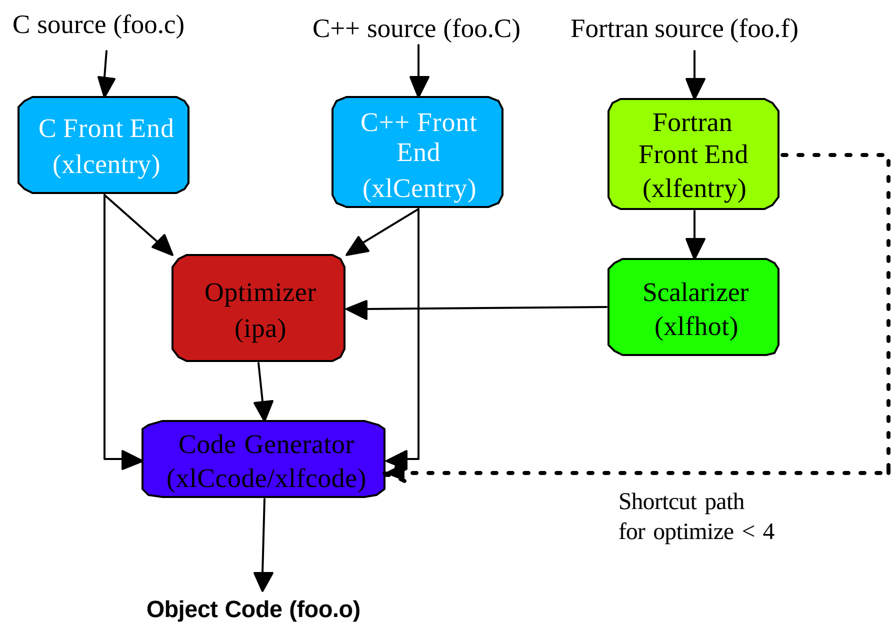
- Use the '[mp]xlf90[_r]' or '[mp]xlf95[_r]' driver invocations where possible to ensure portability. If this is not possible, consider using the -qnosave option.
- When writing new code, use module variables rather than common blocks for global storage.
- Use modules to group related subroutines and functions.
- Use INTENT to describe usage of parameters.
- Limit the use of ALLOCATABLE arrays and POINTER variables to situations which demand dynamic allocation.
- Use CONTAINS only to share thread local storage.
- Avoid the use of -qalias=nostd by obeying Fortran alias rules.
- When using array assignment or WHERE statements, pay close attention to the generated code with -qlist or -qreport. If performance is inadequate, consider using -qhot or rewriting array language in loop form.

C/C++ Programming Tips

- Use the xlc[_r] invocation rather than cc[_r] when possible.
- Always include *string.h* when doing string operations and *math.h* when using the math library.
- Pass large class/struct parameters by address or reference, pass everything else by value where possible.
- Use unions and pointer type-casting only when necessary and try to follow ANSI type rules.
- If a class or struct contains a 'double', consider putting it first in the declaration. If this is not possible, consider using -qalign=natural
- Avoid virtual functions and virtual inheritance unless required for class extensibility. These are costly in object space and function invocation performance.
- Use 'volatile' only for truly shared variables.
- Use 'const' for globals, parameters and functions whenever possible.
- Do limited hand-tuning of small functions by defining them as 'inline' in a header file.

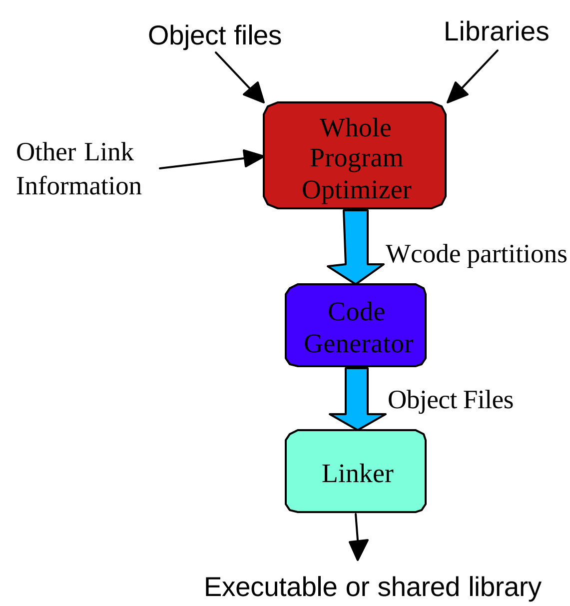
e-business

Inside a Compilation Step



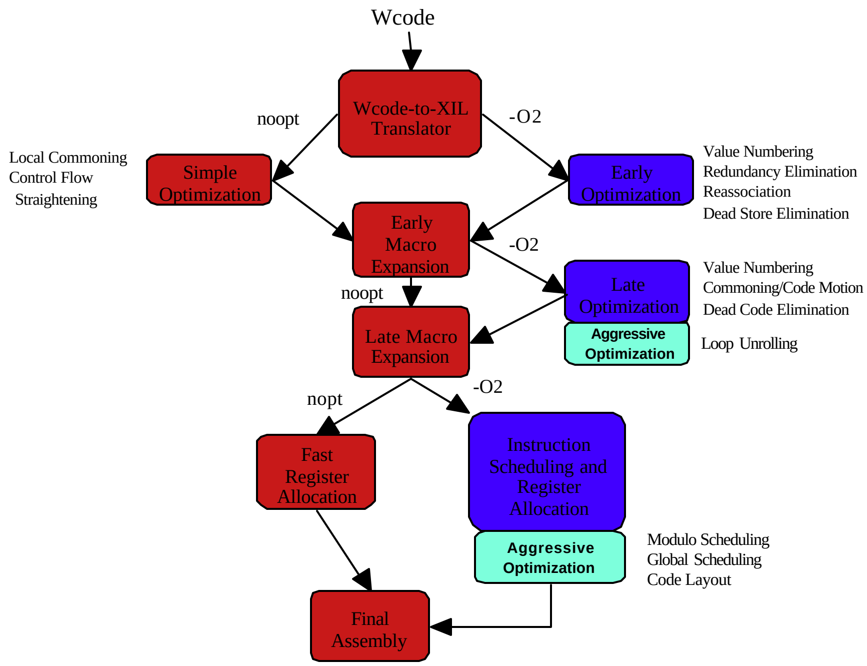
e-business

Inside an Link-time Compilation

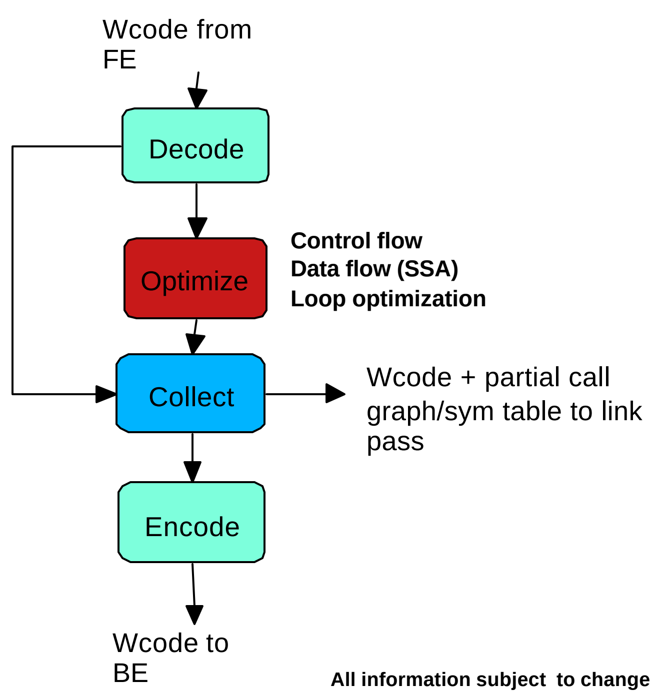


All information subject to change without notice

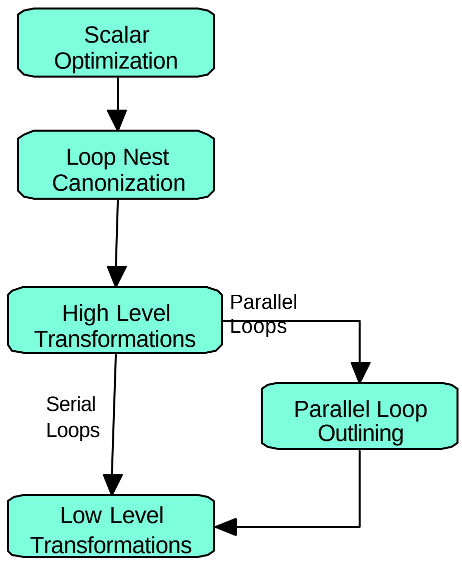
Inside the Code Generator



Inside the Optimizer Compile Pass



Loop Optimization



Control Flow Optimization
Data Flow Optimization
Loop Normalization

Aggressive Copy Propagation
Maximal Loop Fusion

Loop Nest Partitioning
Loop Interchange
Loop Unroll and Jam
Loop Parallelization

Inner Loop Unrolling
Loop Vectorization
Strength Reduction
Redundancy Elimination
Code Motion

OpenMP Example

```
SUBROUTINE SUB(ARR, N, R)
INTEGER N, R
INTEGER ARR(N)

!$OMP PARALLEL DO
REDUCTION(+:R)
DO I=1,N
  ARR(I)=FOO(I,N)
  R=R+BAR(I)
ENDDO
END SUBROUTINE SUB
```

OpenMP Implementation Example

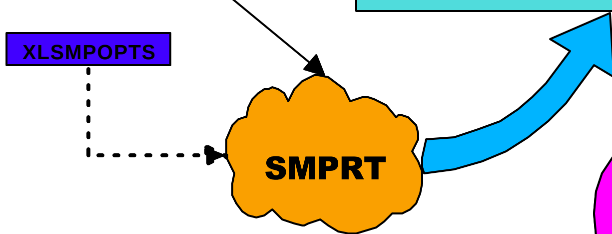
e-business

```
SUBROUTINE SUB(ARR, N,
R)
INTEGER N, R
INTEGER ARR(N)

CALL _xlsmpParDoSetup
(&SUB@OL@1, 1, N)

END SUBROUTINE SUB
```

```
SUBROUTINE SUB@OL@1(FROM, TO)
  INTEGER FROM, TO, I, R1
  R1 = 0
  DO I=FROM, TO
    ARR(I)=FOO(I,N)
    R1=R1+BAR(N)
  ENDDO
  CALL _xlsmpGetLock()
  R=R+R1
  CALL _xlsmpRelLock()
END SUBROUTINE SUB@OL@1
```



Thread management
Data management
Task scheduling
Synchronization

Same mechanism used for automatic parallelism

IBM

All information subject to change without notice

OpenMP Example: XL Fortran V7.1.1 Version

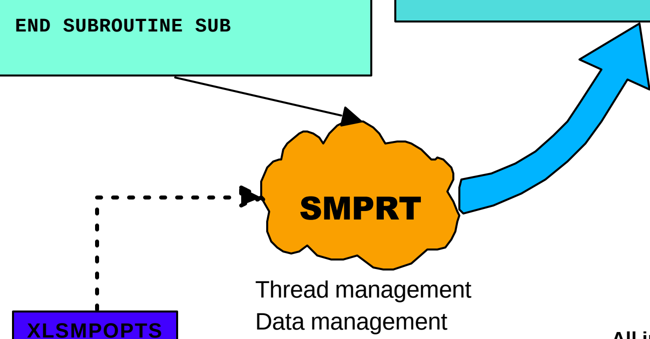
e-business

```
SUBROUTINE SUB(ARR, N, R)
INTEGER N, R
INTEGER ARR(N),RV(32,*)

ALLOCATE (RV(NUM_THREADS))
RV(1,:) = 0
CALL _xlsmpParDoSetup
(&SUB@OL@1, 1, N)
R = SUM(RV(1,:))
DEALLOCATE (RV)

END SUBROUTINE SUB
```

```
SUBROUTINE SUB@OL@1(FROM, TO)
  INTEGER FROM, TO, I, R1
  R1 = RV(1,THREAD_NUM)
  DO I=FROM, TO
    ARR(I)=FOO(I,N)
    R1=R1+BAR(N)
  ENDDO
  RV(1,THREAD_NUM) = R1
END SUBROUTINE SUB@OL@1
```

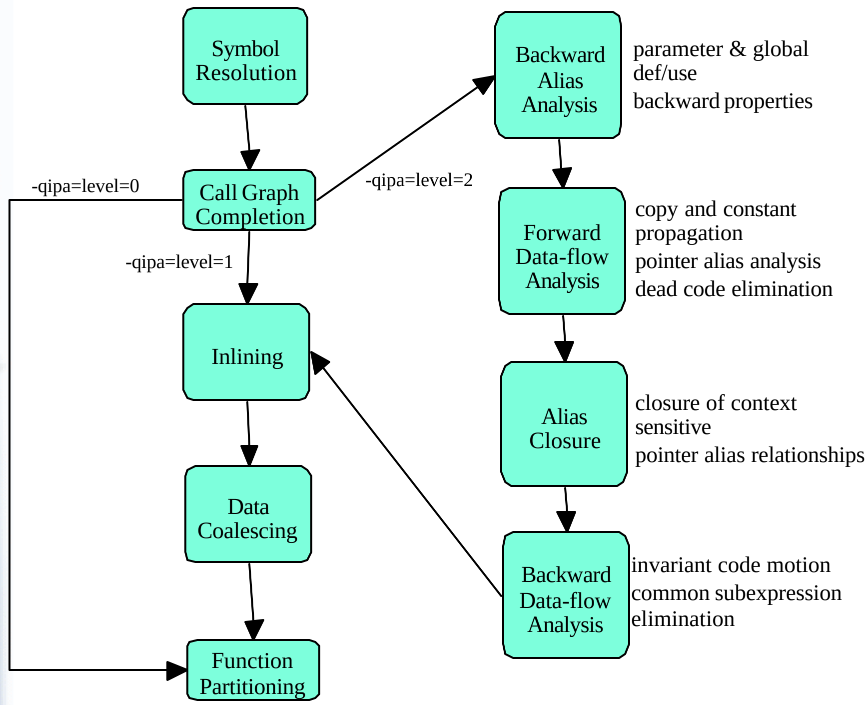


Thread management
Data management
Task scheduling
Synchronization

IBM

All information subject to change without notice

Inside Optimizer Link Pass



Review of Power4 Architecture

- PowerPC 64 bit ISA
- Nominal clock frequency 1.3MHz
- 2 FXUs, 2 FPUs, 2 LSUs, 1 BRU, 1 CRLU per core
- 64K direct I-L1, 32K 2w FIFO D-L1 per core
- 2 cores per chip
- Shared 3x480K 8w PLRU L2 per chip
- Four chips and 4x32MB L3 per module
- 8-32 way configurations

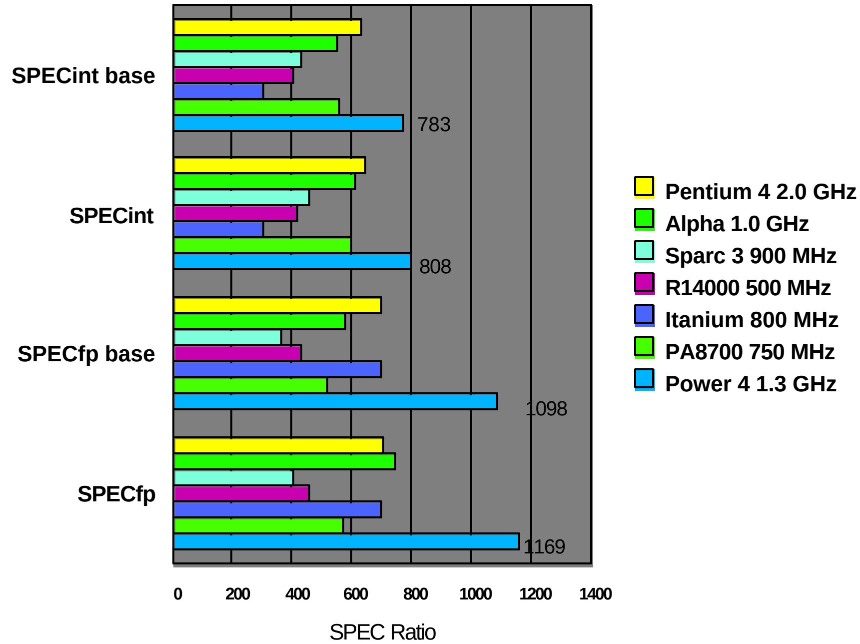
Some interesting Power4 facts

8 instruction fetch buffer
3 cycle pipeline for cracking/preprocessing
4w or 5w (with branch) dispatch with some restrictions
Out-of-order execution, in-order issue and completion
20 entry completion buffer, 1 entry per dispatch group
Renames: 80 GPR, 72 FPR, 24 XER (CA/OV), 16 LR/CTR, 32 CR, 20 FPSCR
2x18 entry FXU/LSU, 2x10 entry FPU instruction queues
Asymmetric FXUs: one does divide, the other SPR ops
2x6 stage LSUs: 2 cycle load-use penalty for FXU, 3 cycle for FPU
8 entry outstanding load miss queue
8 independent data prefetch streams, tracking up or down
2x9 stage FPUs: symmetric, 6 stage execution
1K 4w unified TLB supporting 4K and 16M page sizes

Power 4 Optimization Technology

- **Architecture-neutral and -specific code paths**
  - tuning for arch=ppc and arch=pwr4
- **Precise machine model for scheduling (-O2+)**
  - new instruction scheduler with more detailed modelling capability
  - tuned through extensive experimention on early h/w
- **New loop transformations for deep pipelines (-O3+)**
  - more precise loop unrolling and pipelining
- **New aggressive branch optimizations (-O2+)**
  - branch pattern replacement
  - utilization of branch hints (eg. using profile feedback)
- **Optimized usage of hardware-expanded instructions**
  - eg. load/store update, mtcr, lm/stm
- **Optimized prefetch buffer allocation (-qhot)**
  - utilization of prefetch stream start instructions
  - loop nest fusion and partitioning to optimize # streams

SPEC results for Power4



Competitive data from www.spec.org

Questions?

?